Using = `default` for Special Member Functions

The keyword **`default`** annotating a declaration of a **special member function** instructs the compiler to attempt to generate the function automatically.

Description

An important aspect of C++ class design is the understanding that the compiler will attempt to generate certain member functions to *create*, *copy*, *destroy*, and now *move* (see Section 2.1. "*Rvalue* References" on page 710) an object unless developers implement some or all of these functions themselves. Determining which of the **special member functions** will continue to be generated and which will be suppressed in the presence of **user-provided special member functions** requires remembering the numerous rules the compiler uses.

Declaring a special member function explicitly

The rules specifying what happens in the presence of one or more user-provided special member functions are inherently complex and not necessarily intuitive; in fact, some have been deprecated. Specifically, even in the presence of a user-provided destructor, both the copy constructor and the copy-assignment operator have historically been generated implicitly. Relying on such generated behavior is problematic because it is unlikely that a class requiring a user-provided destructor will function correctly without corresponding user-provided copy operations. As of C++11, reliance on such dubious implicitly generated behavior is deprecated.

Let's briefly illustrate a few common cases and then take a look at Howard Hinnant's now famous table (see page 44 of *Appendix — Implicit Generation of Special Member Functions*) to demystify what's going on under the hood.

Example 1: Providing just the default constructor Consider a **`struct`** with a user-provided default constructor:

```
struct S1
{
    S1();  // user-provided default constructor
};
```

A user-provided default constructor has no effect on other special member functions. Providing any other constructor, however, will suppress automatic generation of the default constructor. We can, however, use **`= default`** to restore the constructor as a **trivial operation**; see *Use Cases — Restoring the generation of a special member function suppressed by another* on page 36. Note that a nondeclared function is nonexistent, which means that it will *not* participate in overload resolution at all. In contrast, a **deleted function** participates